The automatic complexity analysis of divide-and-conquer algorithms

Paul Zimmermann, Wolf Zimmermann

Rapports de Recherche

N°1149

Programme 2

Calcul symbolique, Programmation et Génie logiciel

THE AUTOMATIC COMPLEXITY ANALYSIS OF DIVIDE-AND-CONQUER ALGORITHMS

Paul ZIMMERMANN
Wolf ZIMMERMANN

Décembre 1989

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

The Automatic Complexity Analysis of Divide-and-Conquer Algorithms

Paul Zimmermann [1] and Wolf Zimmermann [2]

Abstract: Current tools performing automatic complexity analysis are capable to deal with function definitions based on structural induction. Divide-and-Conquer Algorithms with "intelligent" divide function (like *quicksort*) are not based on structural induction, but on noetherian induction. This paper presents a method of automatic complexity analysis to deal with such kinds of functions.

L'analyse de complexité en moyenne d'algorithmes de partitionnement récursif

Résumé : Les systèmes actuels d'analyse d'algorithmes reposent sur le principe de l'induction structurelle. Certains algorithmes de partitionnement récursif, comme *quicksort*, sont fondés non pas sur l'induction structurelle, mais sur l'induction noetherienne. Cet article expose une méthode permettant d'analyser automatiquement de tels algorithmes.

[1] INRIA, Rocquencourt, 78153 Le Chesnay (France). This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

[2] Postfach 6980, D-7500 Karlsruhe. The work of this author was partially supported by the ESPRIT-project 510 ToolUse and GMD Karlsruhe.

The Automatic Complexity Analysis of Divide-and-Conquer Algorithms

Paul Zimmermann [‡]
INRIA Rocquencourt

Wolf Zimmermann [§¶]
GMD Karlsruhe

Abstract

Current tools performing automatic complexity analysis are capable to deal with function definitions based on structural induction. Divide-and-Conquer Algorithms with "intelligent" divide function (like e.g. *quicksort*) are not based on structural induction, but on noetherian induction. This paper presents a method of automatic complexity analysis to deal with such kinds of functions.

1 Introduction

Tools for automatic complexity analysis can be used as a support tool for program development tools in order to compare intermediate results of program development tools and control the direction of the development. Hence, automatic complexity analysis is an important area in computer science.

Previous tools (as Metric [Weg75] and ACE [Mét88]) can only analyze function definitions based on structural induction. In order to analyze more practical programs, it is necessary to introduce methods suitable for larger classes of functions. One of these classes contains divide-and-conquer algorithms with an "intelligent" divide function, i.e. where the division of an argument is powerful (e.g. in *quicksort* [Hoa62] the elements less than a certain element, and the elements larger than this element) and conquer is easy (e.g. in *quicksort* just appending the lists). We provide a method of automatic complexity analysis to deal with such kinds of functions.

Two different methods of automatic complexity analysis are known: one is based on recurrences (such as in [Weg75], [Mét88] and [Zim88b]), and the other is based on generating functions (such as in [FSZ89a]).

The first method reduces the problem of analyzing the time complexity of a function to a (possibly conditional) recurrence describing the time behaviour of that function. This recurrence is solved in a second phase. In [Weg75] such a method is proposed for pure LISP programs. Wegbreit analyzes the best case, the worst case, the average case and its variance. In [Mét88] the same is described for FP using FP-algebra. LeMetayer analyzes only the worst case. In [Zim88b], Wegbreit's method is generalized to typed languages.

A more promising approach is the second one, described in [FSZ89a]. The problem of analyzing the average time complexity of a function is reduced to a generating function equation. In a second phase this equation is

[‡] Domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex

[§] Haid–und–Neu–Str 7, 7500 Karlsruhe

[¶] Author's current institution: Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Postfach 6980, 7500 Karlsruhe

solved, an explicit generating function is obtained, and an asymptotic development of its Taylor coefficients is determined.

This paper has to aims: First, we want to show how to analyze divide-and-conquer algorithms. Second we want to compare the two different methods, represented by tools developped at our sides, COMPLEXA [Zim88b] and $\Lambda\Upsilon^{\Omega}$ [FSZ89a].

COMPLEXA is based on the method of recurrences. It is a generalization of [Weg75] to typed programs. The types are defined by equations and represented by a set of constructor terms. Some of the steps described in Section 2 had to be generalized. The symbolic evaluation step uses the data type equations and the mapping onto integers is generalized. For example the *length* of a binary tree is the number of its nodes. The method used in COMPLEXA also contains the method of Section 3. COMPLEXA analyzes the best case, the worst case and the average case complexity of a function.

Lambda-Upsilon-Omega ($\Lambda\Upsilon^{\Omega}$ for short) is a research tool designed to assist the average case analysis of some well defined classes of algorithms and data structures. The $\Lambda\Upsilon^{\Omega}$ system is based on the conjuction of recent methodologies in combinatorial analysis, and powerful theorems of complex analysis. The *Algebraic Analyzer* translates an algorithm and its data structures into generating functions. The *Analytic Analyzer* extracts the *asymptotic form* of coefficients of generating functions. When one puts these two subsystems together, one gets a system ($\Lambda\Upsilon^{\Omega}$) that gives an asymptotic development for the average cost of an algorithm. In the current stage, $\Lambda\Upsilon^{\Omega}$ can analyze symbolic differentiation algorithms such as formal differentiation, simplification and rewriting systems, or some parameters over combinatorial structures such as functional graphs or various trees.

We start with the method of recurrences, which is based on analyzing dependencies at the level of recurrences. We show that the analysis of divide-and-conquer algorithms leads to a *family of recurrences*, and how to deal with them. The second part of this paper explains the second method. It will be shown that for divide-and-conquer algorithms, *differential equations* instead of algebraic equations are obtained for the generating function. Finally we compare the two methods in general.

The paper is organised as follows: In Section 2, Wegbreit's method is briefly introduced. In Section 3 we extend the method to divide-and-conquer algorithms. In Section 4, the method based on generating functions is described and in Section 4.3 its extension to quicksort is detailed. Finally, we compare the results of both methods and give an outlook of further directions of work.

2 Wegbreit's Method

Wegbreit's method [Weg75] is based on recurrences. The first step is to find a (possibly conditional) recurrence, describing the time behaviour of a function. Then these recurrences are solved. Conditional recurrences are dealt by best/worst case analysis, and when averaging by using probabilities for conditions to be true. The main features are demonstrated by the following example.

Example 2.1: (`append` of two lists)

```
append(l,m) =
   empty(l) -> m
   cons(car(l),append(cdr(l),m))
```

The task is to analyze the time complexity of this function. For simplicity we assume in the whole paper, that each basic function (*empty, nil, cons, car, cdr*) and basic functional (reference to a variable, decision in a conditional, function call) needs 1 time unit. In principle it is possible to deal with symbolic values as proposed in [Weg75].

Step 1: Transform the function to a function which computes the time complexity. In the example this step yields:

$$
\begin{array}{l}
time(\texttt{append(l,m)}) = \\
\quad \texttt{empty(l)} \rightarrow 4 \\
\quad 10 + time(\texttt{append}(\texttt{cdr}(\texttt{l}), \texttt{m}))
\end{array}
$$

Step 2: Derive equations describing inductively the function obtained by the first step. This step is divided into three substeps. The first is normalisation removing nested conditionals. After this step there is no conditional $(c \rightarrow t; e)$ where the 'then-expression' t is itself a conditional. In the example, no such normalisation is necessary. The second substep removes irrelevant argument positions. In our example the second argument position is irrelevant, because m does not change during recursion. Thus we remove it and get:

$$
\begin{array}{l}
time(\texttt{append(l)}) = \\
\quad \texttt{empty(l)} \rightarrow 4 \\
\quad 10 + time(\texttt{append}(\texttt{cdr}(\texttt{l})))
\end{array}
$$

The third substep is a symbolic evaluation step, which finds the inductive equations describing the result obtained in the second substep. In the example we obtain:

$$
\begin{array}{rcl}
E(nil) & = & 4 \\
E(cons(a, l)) & = & 10 + E(l)
\end{array}
$$

where $E(l) = time(\texttt{append(l)})$

Step 3: The equations obtained by the second step are transformed to a recurrence by mapping lists onto integers. Two such mappings are provided:

$$
\begin{array}{rcl@{\qquad}rcl}
length(nil) & = & 0 & size(nil) & = & 0 \\
length(cons(a, l)) & = & 1 + length(l) & size(cons(a, l)) & = & 1 + size(a) + size(l)
\end{array}
$$

In the example, *length* is chosen, because a does not occur on the RHS of the second equation. With $n = length(l)$ and $a_n = E(l)$, we obtain the recurrence:

$$
\begin{array}{rcl}
a_0 & = & 4 \\
a_{n+1} & = & 10 + a_n
\end{array}
$$

Step 4: Solving the recurrence. The above recurrence yields

$$a_n = 4 + 10 \cdot n$$

Thus, the time complexity of `append` is:

$$time(\texttt{append(l,m)}) = 4 + 10 \cdot length(\texttt{l})$$

It is easy to solve all linear recurrences with constant coefficients and polynomial or exponential inhomogeneities [Hen77, GKP89].

We describe now the deficiencies of that method. Let *eval* be an operational semantics of pureLISP. Wegbreit deals the function composition as follows:

$$time(f(g(x)) = time(g(x)) + time(f(eval[g(x)])) + 1$$

For evaluating the second time-expression, it is necessary to analyze first $time(f(y))$. Let the result of this analysis be $h(M(y))$, where $M \in \{length, size\}$. Thus $M(eval[g(x)])$ must be analyzed, which is done in almost the same way as the time analysis. For the details see [Weg75]. If $M(eval[g(x)])$ is *deterministic*, i.e. its best case, its worst case, and its average case are all the same, then the function composition is dealt completely. On the other side, if $M(eval[g(x)])$ is *probabilistic* (i.e. not deterministic), then this method is not applicable. Exactly this problem occurs in the analysis of divide-and-conquer algorithms such as *quicksort*. In the next section we show how to tackle this problem.

3 Divide-and-Conquer Algorithms

The method is based on the observation, that if $M(eval[g(x)])$ is probabilistic, then its corresponding recurrence had to be a conditional recurrence, and on the other side, that two arguments in the body of divide-and-conquer functions can depend on each other (e.g. in *quicksort* the arguments are two lists which consist of elements of the original argument and are disjoint).

Therefore, the previous method must be changed in the following way: First, each argument is reduced to recurrences. Second, if two of these recurrences are conditional recurrences with the *same* conditions, then a dependence analysis at the level of recurrences is performed. We demonstrate this method by the example of *quicksort* which is the fastest known sorting algorithm introduced by [Hoa62]. We will see that the results of the automatic analysis fit the well known results in [AHU74] and [Knu73].

Example 3.1: (*quicksort*)

```
quicksort(l) =
   empty(l) -> nil
   append(quicksort(selle(car(l),cdr(l))),
          cons(car(l),quicksort(selgt(car(l),cdr(l)))))

append(l,m) =
   empty(l) -> m
   cons(car(l),append(cdr(l),m))

selle(a,l) =
   empty(l) -> nil
   car(l) <= a -> cons(car(l),selle(a,cdr(l)))
   selle(a,cdr(l))

selgt(a,l) =
   empty(l) -> nil
   car(l) <= a -> selgt(a,cdr(l))
   cons(car(l),selgt(a,cdr(l)))
```

3.1 Obtaining a Family of Recurrences

The first step in the analysis of example 3.1 is to analyze in parallel $time($`append(l,m)`$)$, $time($`selle(a,l)`$)$ and $time($`selgt(a,l)`$)$, i.e. we first determine all the recurrences.

The analysis of $time($`append(l,m)`$)$ leads to an unconditional recurrence. This kind of recurrences can be tackled as in Section 2 and be solved immediately yielding:

$$time(\texttt{append(l,m)}) = 4 + 10 \cdot length(\texttt{l})$$

The second analysis, $time(\texttt{selle(a,l)})$ reduces to the equations:

$$\begin{aligned} E(nil) &= 4 \\ E(cons(b,l)) &= \begin{cases} 15 + E(l) & \text{if } b \leq a \\ 12 + E(l) & \text{otherwise} \end{cases} \end{aligned}$$

where $E(l) = time(\texttt{selle(a,l)})$. With $n = length(l)$ and $a_n = E(l)$ we get the conditional recurrence

$$\begin{aligned} a_0 &= 4 \\ a_{n+1} &= \begin{cases} 15 + a_n & \text{if } b \leq a \\ 12 + a_n & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

Similarly, from $time(\texttt{selgt(a,l)})$ is obtained:

$$\begin{aligned} b_0 &= 4 \\ b_{n+1} &= \begin{cases} 12 + b_n & \text{if } b \leq a \\ 15 + b_n & \text{otherwise} \end{cases} \end{aligned} \tag{2}$$

The goal is now to find r_n and s_n such that $r_n \cdot a_n + s_n \cdot b_n$ is independent of the condition $b \leq a$. This can be computed by the following theorem:

Theorem 3.2: (Sufficient Condition for Dependence)
Let *cond* be any condition and

$$\begin{aligned} a_0 &= c_0 \\ a_{n+1} &= \begin{cases} p_1(n+1) + c_1(n) \cdot a_n & \text{if } cond \\ p_2(n+1) + c_1(n) \cdot a_n & \text{otherwise} \end{cases} \end{aligned} \tag{3}$$

$$\begin{aligned} b_0 &= d_0 \\ b_{n+1} &= \begin{cases} q_1(n+1) + d_1(n) \cdot b_n & \text{if } cond \\ q_2(n+1) + d_2(n) \cdot b_n & \text{otherwise} \end{cases} \end{aligned} \tag{4}$$

be two conditional recurrences. If

$$d_1(n)(p_1(n+1) - p_2(n+1))(q_2(n) - q_1(n)) = c_1(n)(q_2(n+1) - q_1(n+1))(p_1(n) - p_2(n)) \tag{5}$$

then

$$c_n = (q_2(n) - q_1(n))a_n + (p_1(n) - p_2(n))b_n$$

is independent of *cond* and can be computed recursively.

Proof:
The sequence c_{n+1} can be obtained by formal addition of the two equations (3) and (4) according to the definition of c_n. This yields

$$\begin{aligned} c_{n+1} &= \begin{cases} p_1(n+1)\ q_2(n+1) - p_2(n+1)\ q_1(n+1) \\ c_1(n)\ a_n\ ((q_2(n+1) - q_1(n+1)) + d_1\ b_n\ (p_1(n+1) - p_2(n+1)) & \text{if } cond \\ \\ -p_2(n+1)\ q_1(n+1) + p_1(n+1)\ q_2(n+1) \\ c_1(n)\ a_n\ ((q_2(n+1) - q_1(n+1)) + d_1\ b_n\ (p_1(n+1) - p_2(n+1)) & \text{otherwise} \end{cases} \\ &= p_1(n+1)\ q_2(n+1) - p_2(n+1)\ q_1(n+1) + \\ &\quad \frac{q_2(n+1) - q_1(n+1)}{q_2(n) - q_1(n)} c_1(n)\ (q_2(n) - q_1(n))a_n + \\ &\quad \frac{p_1(n+1) - p_2(n+1)}{p_1(n) - p_2(n)} d_1(n)\ (p_1(n) - p_2(n))b_n \\ &= p_1(n+1)\ q_2(n+1) - p_2(n+1)\ q_1(n+1) + h(n)\ c_n \end{aligned} \tag{6}$$

where

$$h(n) = \frac{q_2(n+1) - q_1(n+1)}{q_2(n) - q_1(n)} c_1(n) = \frac{p_1(n+1) - p_2(n+1)}{p_1(n) - p_2(n)} d_1(n)$$

because of equation (5).

Thus we have an unconditional recurrence for c_n which yields a solution independent whether the *cond* is true or false at any stage. □

In the example, $r_n = s_n = 1$ is obtained, which leads by application of (1), (2) the recurrence:

$$\begin{aligned} (a_0 + b_0) &= 8 \\ (a_{n+1} + b_{n+1}) &= 27 + (a_n + b_n) \end{aligned}$$

The solution of this recurrence is

$$a_n + b_n = 8 + 27 \cdot n$$

and we obtain:

$$time(\texttt{selle(a,l)}) + time(\texttt{selgt(a,l)}) = 8 + 27 \cdot length(\texttt{l})$$

Using these results, we get the following equations for $E(l) = time(\texttt{quicksort(l)})$:

$$\begin{aligned} E(nil) &= 4 \\ E(cons(a,l)) &= 28 + 10 \cdot length(eval[\texttt{quicksort(selle(a,l))}]) + \\ &\quad 27 \cdot length(\texttt{l}) + E(\texttt{selle(a,l)}) + E(\texttt{selgt(a,l)}) \end{aligned} \tag{7}$$

For deriving the recurrences, we choose the abbreviations $n = length(l)$ and $a_n = E(l)$. The expressions $length(eval\ [\texttt{quicksort(selle(a,l))}])$, $length(eval\ [\texttt{selle(a,l)}])$ and $length(eval\ [\texttt{selgt(a,l)}])$ must be analyzed in parallel. Using the result:

$$length(\texttt{quicksort(l)}) = length(\texttt{l})$$

we obtain from (7) the equations

$$\begin{aligned} E(nil) &= 4 \\ E(cons(a,l) &= 28 + 10 \cdot length(eval[\texttt{selle(a,l)}]) + \\ &\quad +27 \cdot length(\texttt{l}) + E(\texttt{selle(a,l)}) + E(\texttt{selgt(a,l)}) \end{aligned} \tag{8}$$

Now, analyzing $length(eval[\texttt{selle(a,l)}])$ yields with the abbreviation $E(l) = length(eval[\texttt{selle(a,l)}])$:

$$\begin{aligned} E(nil) &= 0 \\ E(cons(b,l)) &= \begin{cases} 1 + E(l) & \text{if } b \le a \\ E(l) & \text{otherwise} \end{cases} \end{aligned}$$

This leads with $n = length(l),\ a_n = E(l)$ to

$$\begin{aligned} a_0 &= 0 \\ a_{n+1} &= \begin{cases} 1 + a_n & \text{if } b \le a \\ a_n & \text{otherwise} \end{cases} \end{aligned}$$

Similarly, $length(eval[\texttt{selgt(a,l)}]) = b_n$ where $n = length(l)$ leads to

$$\begin{aligned} b_0 &= 0 \\ b_{n+1} &= \begin{cases} b_n & \text{if } b \le a \\ 1 + b_n & \text{otherwise} \end{cases} \end{aligned}$$

Theorem 3.2 shows that $a_n + b_n$ is independent of n and yields the recurrence:

$$\begin{aligned} (a_0 + b_0) &= 0 \\ (a_{n+1} + b_{n+1}) &= 1 + (a_n + b_n) \end{aligned}$$

which has the solution $a_n + b_n = n$. Thus we know that

$$length(eval[\texttt{selle(a, l)}]) + length(eval[\texttt{selgt(a, l)}]) = length(\texttt{l})$$

Setting $n = length(\texttt{l})$ and $i = length(eval[\texttt{selle(a, l)}])$, we obtain from (8) the *family* of recurrences:

$$\begin{aligned} a_0 &= 4 \\ a_{n+1} &= 28 + 10 \cdot i + 27 \cdot n + a_i + a_{n-i}, \qquad 0 \leq i \leq n \end{aligned}$$

The result

$$length(\texttt{quicksort(l)}) = length(\texttt{l})$$

used in the example can be obtained in a similar way. Now, it must be explained how to deal with families of recurrences:

3.2 Families of Recurrences

We have to analyze all $n + 1$ solutions for their best case, their worst case and their average case. We demonstrate this kind of analysis by the family of recurrences obtained in Section 3.1

Best Case Analysis: The minimum occurs if either $28 + 10\ i = 27\ n$ is as small as possible, or minimizing in parallel i and $n - i$. first case $i = 0$ must be chosen, which leads to the recurrence

$$\begin{aligned} a_0 &= 4 \\ a_{n+1} &= 32 + 27\ n + a_n \end{aligned}$$

This recurrence has the solution:

$$a_n = 4 + 45.5\ n + 13.5\ n^2$$

For the second case, the approximation $i = \frac{n}{2}$ is chosen. We consider the approximate recurrence:

$$\begin{aligned} a_1 &= 36 \\ a_n &= 28 + 32 \cdot n + 2 \cdot a_{\frac{n}{2}} \end{aligned}$$

which has the solution

$$a_n = 32\ n\ \log_2 n + O(n)$$

For large n, this solution is less than the above result. Thus the best case of quicksort is:

$$time(\texttt{quicksort(l)}) = 32 \cdot length(\texttt{l}) \cdot \log_2 length(\texttt{l}) + O(length(\texttt{l}))$$

Worst Case Analysis: The maximum occurs if $28 + 10\ i + 27\ n$ is as large as possible and one of i or $n - i$ is as large as possible. This process yields $i = n$ and results in the recurrence

$$\begin{aligned} a_0 &= 4 \\ a_{n+1} &= 32 + 37\ n + a_n \end{aligned}$$

which has the solution $a_n = 4 + 40.5\ n + 18.5\ n^2$. Thus, the worst case is:

$$time(\texttt{quicksort(l)}) = 18.5 \cdot length^2(\texttt{l}) + 40.5 \cdot length(\texttt{l}) + 4$$

Average Case Analysis: The average case analysis is performed under uniform distribution assumption, i.e. each $0 \leq i \leq n$ occurs with the same probability $\frac{1}{n+1}$. This assumption is justified, if each permutation

of list with n elements occurs with the same probability (see [Hen89]). Thus we can average over all choices of i and get

$$\begin{aligned} a_0 &= 4 \\ a_{n+1} &= 28 + 32\ n + \frac{2}{n+1}\ \sum_{i=0}^{n} a_i \end{aligned}$$

The last equation is equivalent to

$$(n+1)\ a_{n+1} = (n+1)\ (28 + 32\ n) + 2\ \sum_{i=0}^{n} a_i$$

Subtracting from this equation

$$n\ a_n = n\ (32\ n - 4) + 2\ \sum_{i=0}^{n-1} a_i$$

yields

$$(n+1)\ a_{n+1} - n\ a_n = 64\ n + 28 + 2\ a_n$$

which leads to the recurrence

$$\begin{aligned} a_0 &= 4 \\ a_{n+1} &= 64 - \frac{36}{n+1} + \frac{n+2}{n+1}\ a_n \end{aligned}$$

This recurrence can be solved by direct summing:

$$\begin{aligned} a_n &= \sum_{i=1}^{n} \left(\prod_{j=i}^{n-1} \frac{j+2}{j+1} \right) \left(64 - \frac{36}{i+1} \right) + 4\ (n+1) \\ &= 64\ n\ H_{n+1} - 96\ n + 64\ H_{n+1} - 60 \end{aligned}$$

where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the n-th harmonic number.

Because of the expansion $H_n = \log n + \gamma + \frac{1}{2n} + O(1/n^2)$, we replace H_n by this expression and get the average time complexity of quicksort:

$$\begin{aligned} &time(\texttt{quicksort(l)}) = \\ &\quad 64 \cdot n \cdot \log n + (64\gamma - 96) \cdot n + 64 \cdot \log n + 64 \cdot \gamma + 36 + O(1/n) \end{aligned}$$

where $n = length(\texttt{l})$.

4 Generating Functions and Structural Induction

The main idea of this method is that recurrences can also be represented by generating functions. For example, we associate to the recurrence for Fibonacci numbers

$$F_0 = 0 \quad F_1 = 1 \qquad F_{n+2} = F_{n+1} + F_n \quad n \geq 0$$

the following generating function

$$F(z) = \sum_{n=0}^{\infty} F_n z^n = \frac{z}{1 - z - z^2}$$

The fact that there exists a closed form for $F(z)$ is strongly related to the linearity of the above recurrence. We will show below that one can derive *directly* generating functions from data type or procedure declarations. This method is particularly well suited for asymptotic analysis: theorems of singularity analysis, like those of Jüngen and Flajolet-Odlyzko ([Sal88]), give an expansion of $[z^n]F(z)$[1] when n tends to infinity.

4.1 From generating functions to asymptotic cost

To compute the average complexity of a program P, whose inputs are equally distributed over a set A, the first step computes the *counting generating function* of A

$$A(z) = \sum_{a \in A} z^{|a|}$$

where $|a|$ is the size of the element a. The second step computes the *complexity descriptor* of P

$$\tau P(z) = \sum_{a \in A} \tau P(a) z^{|a|}$$

where $\tau P(a)$ is the cost of the execution of P when we take a as input. The number $[z^n]\tau P(z)$ is the total cost of P over all inputs of size n. Whence the average complexity of P over all inputs of size n is

$$\overline{\tau P_n} = \frac{[z^n]\tau P(z)}{[z^n]A(z)}. \tag{9}$$

This formula shows that if we know an asymptotic expansion for $[z^n]\tau P(z)$ and for $[z^n]A(z)$, we can divide them to obtain an asymptotic expansion for the average complexity of P.

4.2 From program to generating functions

For algorithms operating over *decomposable* data structures, some well defined rules allow to translate program schemes into operators over generating functions. The simplest example is the *sequencing scheme*

```
procedure R (a : A);
begin
  P(a) ; Q(a)
end;
```

which translates into the sum operator

$$\tau R(z) = \tau P(z) + \tau Q(z).$$

An other example is the *component descent*: if C is the Cartesian product of A and B, the declaration

```
procedure R (c : C);
case c of
  (a,b) : P(a);
end;
```

translates into

$$\tau R(z) = \tau P(z) b(z).$$

There are also similar rules for the translation from types constructors (union, Cartesian product, sequence of, set of, cycle of) to counting generating functions. These rules are described in [Zim88a].

[1] the notation $[z^n]f(z)$ is used to denote the coefficient of z^n in the Taylor expansion of $f(z)$

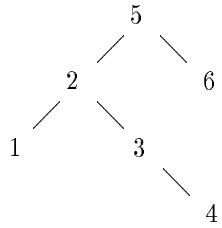
4.3 Application to Quicksort

In this section, we first show that analyzing quicksort reduces to the analysis of some function over *heap ordered trees*. Then we detail all steps that allow to compute automatically generating functions and complexity descriptors. Finally we show the corresponding $\Lambda\Upsilon^{\Omega}$ session.

Binary search trees and heap ordered trees The *binary search tree* of a sequence S is

$$\mathcal{BST}(S) = \begin{cases} (\mathcal{BST}(S_{\leq}), s_1, \mathcal{BST}(S_{>})) & \text{if } |S| \geq 1 \\ \Box & \text{if } |S| = 0 \end{cases}$$

where s_1 is the first element in S, $S_{\leq}$ is the subsequence of elements $\leq s_1$, and $S_{>}$ is the subsequence of elements $> s_1$. An empty binary search tree is represented by the external node $\Box$. For example, the binary search tree of $(5, 2, 6, 3, 4, 1)$ is (we do not draw external nodes)



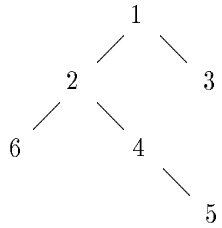
The *heap ordered tree* of a sequence S is

$$\mathcal{HOT}(S) = \begin{cases} (\mathcal{HOT}(S_{left}), min(S), \mathcal{HOT}(S_{right})) & \text{if } |S| \geq 1 \\ \Box & \text{if } |S| = 0 \end{cases}$$

where $min(S)$ is the rightmost smallest element in S and S_{left} (resp. S_{right}) is the subsequence of elements to the left (resp. right) of $min(S)$. W. Burge (see [Bur76] or [Vui80]) showed the following equivalence principle:

> *For each pair of inverse permutations σ and σ^{-1}, we have*
>
> $$\mathcal{BST}(\sigma) \equiv_{shape} \mathcal{HOT}(\sigma^{-1})$$
>
> *where $t \equiv_{shape} u$ means that the unlabeled trees associated to t and u are identical.*

The inverse permutation of $(5, 2, 6, 3, 4, 1)$ is $(6, 2, 4, 5, 1, 3)$, whose heap ordered tree is



As expected, it has the same shape as the binary search tree of $(5, 2, 6, 3, 4, 1)$.
Now let us apply this to quicksort. Our aim is to determine the average cost of quicksort over all permutations of size n. Let us take the same cost definition as given in Section 3.1:

$$\begin{aligned} time(\text{quicksort}(())) &= 4 \\ time(\text{quicksort}(cons(s_1, l))) &= 31 + 10\, length(\text{quicksort}(\text{selle}(s_1, l))) + 27\, length(l) \\ &+ time(\text{selle}(s_1, l)) + time(\text{selgt}(s_1, l)) \end{aligned}$$

We now apply this to the binary search tree of $S = cons(s_1, l)$:

$$\begin{aligned} time(\text{quicksort}(\Box)) &= 4 \\ time(\text{quicksort}(B_{\leq}, s_1, B_{>})) &= 31 + 10\, size(B_{\leq}) + 27\, (size(B_{\leq}) + size(B_{>})) \\ &+ time(\text{quicksort}(B_{\leq})) + time(\text{quicksort}(B_{>})). \end{aligned}$$

The above system only depends on the *shape* of the binary search trees, thus we can replace them by heap ordered trees, without changing the total cost over all data structures of size n.

$$\begin{aligned} time(\text{quicksort}(\square)) &= 4 \\ time(\text{quicksort}(H_{\text{left}}, s_1, H_{\text{right}})) &= 31 + 10\, size(H_{\text{left}}) + 27\,(size(H_{\text{left}}) + size(H_{\text{right}})) \\ &+ time(\text{quicksort}(H_{\text{left}})) + time(\text{quicksort}(H_{\text{right}})). \end{aligned}$$

Data type analysis We have here *labelled* objects, thus we use *exponential* generating functions instead of *ordinary* generating functions. The exponential generating function of a class A is

$$\hat{A}(z) = \sum_{a \in A} \frac{z^{|a|}}{|a|!} = \sum_{n=0}^{\infty} A_n \frac{z^n}{n!}$$

where A_n is the number of elements of size n in A. The rules we need here are

$$\begin{aligned} A = \text{Latom}(k) &\Rightarrow \hat{A}(z) = z^k/k! \\ A = B \cup C &\Rightarrow \hat{A}(z) = \hat{B}(z) + \hat{C}(z) \\ A = B \times C &\Rightarrow \hat{A}(z) = \hat{B}(z)\hat{C}(z) \\ A = min(B) \times C &\Rightarrow \hat{A}(z) = \int \hat{B}'(z)\hat{C}(z)dz \end{aligned}$$

We obtain thus the following system

$$\begin{aligned} \hat{H}(z) &= 1 + \int \hat{H}(z) key'(z) \hat{H}(z) dz \\ key(z) &= z \end{aligned}$$

Replacing $key(z)$ by z in the first equation gives $\hat{H}(z) = 1 + \int \hat{H}^2(z)dz$ whose solution is $\hat{H}(z) = 1/(C - z)$, for some constant C. The number of heap ordered trees of size 0 is 1, hence $C = 1$ and

$$\hat{H}(z) = \frac{1}{1-z}$$

Procedure analysis As for data structures, we will use here *exponential* complexity descriptors. The rules for procedures on labelled objects are

`procedure R(a:A); begin P(a);Q(a) end;`	$\Rightarrow$	$\hat{\tau}R(z) = \hat{\tau}P(z) + \hat{\tau}Q(z)$
`procedure P(a:A); count; measure count:1;`	$\Rightarrow$	$\hat{\tau}P(z) = a(z)$
`type A = min(B) C; procedure R(a:A); case a of (b,c) : P(c) end;`	$\Rightarrow$	$\hat{\tau}R(z) = \int b'(z)\hat{\tau}P(z)dz$
`type A = min(B) C; procedure R(a:A); case a of (b,c) : P(b) end;`	$\Rightarrow$	$\hat{\tau}R(z) = \int \hat{\tau}'P(z)c(z)dz$

Applying these rules to the procedure `size` gives:

$$\hat{\tau}\text{size}(z) = \int \hat{H}^2(z)dz + 2\int \hat{\tau}\text{size}(z)\hat{H}(z)dz$$

After substitution of $\hat{H}(z)$ by $1/(1-z)$, derivation and resolution, we obtain $\hat{\tau}\text{size}(z) = (z + C)/(1 - z)^2$ for some constant C. The total cost of `size` over trees of size 0 is 0, hence $size(0) = C = 0$

$$\hat{\tau}\text{size}(z) = \frac{z}{(1-z)^2}$$

Applying the above rules to the procedure `quicksort` leads to

$$\hat{\tau}\text{quicksort}(z) = 4 + 31\int \hat{H}^2(z)dz + 64\int \hat{\tau}\text{size}(z)\hat{H}(z)dz + 2\int \hat{\tau}\text{quicksort}(z)\hat{H}(z)dz$$

After substitution of $\hat{H}(z)$ and $\tau\text{size}(z)$, derivation and resolution, we get $\hat{\tau}\text{quicksort}(z) = (64\log(1/(1 - z)) - 33z + C)/(1 - 2z + z^2)$ for some constant C. The total cost of `quicksort` over trees of size 0 is 4, hence $C = 4$ and

$$\hat{\tau}\text{quicksort}(z) = \frac{64\log(1/(1-z)) - 33z + 4}{1 - 2z + z^2}$$

Asymptotic analysis The formula (9) applied to `quicksort` leads to

$$\overline{\tau\text{quicksort}}_n = \frac{[z^n]\hat{\tau}\text{quicksort}(z)}{[z^n]\hat{H}(z)}$$

The coefficient of z^n in the Taylor expansion of $\hat{H}(z) = 1/(1-z)$ is simply 1. The coefficient of z^n in the Taylor expansion of τquicksort(z) is computed automatically by the program `equivalent` (cf [Sal88]), which gives

$$[z^n]\tau\text{quicksort}(z) = 64n\log n + (64\gamma - 93)n + 64\log n + 64\gamma + 36 + O(\frac{1}{n^2})$$

using Jüngen's theorem.

5 Concluding remarks

In this section, we compare both methods and give some ideas of further research. As shown in figure 1, both methods contain three main steps: they first derive equations for the complexity of the algorithm, then these equations are solved, and finally an asymptotic expansion is computed.

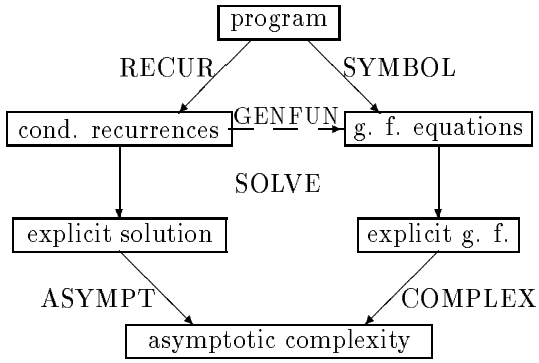


Figure 1: From program to asymptotic complexity via recurrences or generating functions

The extension of Wegbreit's method proposed here enables to analyze some schemes of *functional composition* that lead to *conditional recurrences*. Other systems like Metric, ACE and $\Lambda\Upsilon\Omega$ could not deal with such kind of functional composition, because it is not decidable in the general case. This method also gives the best and worst case analysis, and can apply directly to algorithms with an "intelligent" divide function like quicksort, whereas such algorithms must be rewritten using some meta–knowledge to be analyzed by $\Lambda\Upsilon\Omega$.
Tools for asymptotic analysis are currently more developed for generating functions than for recurrences. This is a great advantage of the generating function method, together with the fact that generating functions are easy to handle with a computer algebra system. Another new result is the extension of Greene's box operator to procedures, which leads to differential equations over complexity descriptors.
Further research includes the resolution of recurrences via generating functions (arrow $GENFUN$ in figure 1), which would give asymptotic results for some recurrences that have unpleasant explicit solutions (for example the solution of $a_n = a_{n-1} + a_{n-2} + H_n$ where H_n is the n-th harmonic number). Another research field is to extend the analytic analyzer (arrow $COMPLEX$) to equations that can not be solved explicitly (this includes functional equations such as $f(z) = 1 + zf(z^2 + z^3)$).
In summary, we expect that our work will bring new motivations in the field of automatic complexity analysis.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Bur76] W. H. Burge. An analysis of binary search trees formed from sequences of nondistinct keys. *Journal of the ACM*, 23(3):451–454, July 1976.

[CGG+88] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt. *MAPLE: Reference Manual*. University of Waterloo, 1988. 5th edition.

[FSZ89a] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda–Upsilon–Omega: An Assistant Algorithms Analyzer. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 201–212, June 1989.

[FSZ89b] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda–Upsilon–Omega: The 1989 Cookbook. Rapport de recherche 1073, Institut National de Recherche en Informatique et en Automatique, August 1989. 116 pages.

[GKP89] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison Wesley, 1989.

[Hen77] P. Henrici. *Applied and Computational Complex Analysis*. John Wiley, New York, 1977.

[Hen89] P. Hennequin. Combinatorial Analysis of Quicksort Algorithm. *Theoretical Informatics and Applications*, 23(3):317–333, 1989.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3 : Sorting and Searching. Addison-Wesley, 1973.

[Mét88] D. Le Métayer. Ace: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[Sal88] B. Salvy. Fonctions génératrices et asymptotique automatique. Rapport de DEA, Université Paris XI, 1988. Also available as INRIA Research Report 967 (1989).

[Vui80] J. Vuillemin. A Unifying Look at Data Structures. *Communications of the ACM*, 23(4):229–239, April 1980.

[Weg75] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

[Zim88a] P. Zimmermann. Alas : un système d'analyse algébrique. Rapport de DEA, Université de Paris VII, 1988. 120 pages. Disponible aussi en rapport de recherche INRIA (numéro 968).

[Zim88b] W. Zimmermann. How to Mechanize Complexity Analysis. Technical report, Karlsruhe, 1988.